DOI:
10.1007/s00224-017-9760-2

Document Version
Peer reviewed version

Theory Comput Syst (2017) 61:606–636
DOI 10.1007/s00224-017-9760-2

Optimizing Binary Heaps

Stefan Edelkamp[1] · **Amr Elmasry**[2] · **Jyrki Katajainen**[3]

Published online: 21 April 2017

Abstract A priority queue—a data structure supporting, inter alia, the operations *minimum* (*top*), *insert* (*push*), and *extract-min* (*pop*)—is said to operate in-place if it uses $O(1)$ extra space in addition to the n elements stored at the beginning of an array. Prior to this work, no in-place priority queue was known to provide worst-case guarantees on the number of element comparisons that are optimal up to additive constant terms for both *insert* and *extract-min*. In particular, for the standard implementation of binary heaps, *insert* and *extract-min* operate in logarithmic time while involving at most $\lceil \lg n \rceil$ and $2 \lg n$ [could possibly be reduced to $\lg \lg n + O(1)$ and $\lg n + \log^* n + O(1)$] element comparisons, respectively. In this paper we propose a variant of a binary heap that operates in-place, executes *minimum* and *insert* in $O(1)$ worst-case time, and *extract-min* in $O(\lg n)$ worst-case time while involving at most $\lg n + O(1)$ element comparisons. These efficiencies surpass lower bounds known for binary heaps, thereby resolving a long-standing theoretical debate.

A preliminary version of this paper entitled "An in-place priority queue with $O(1)$ time for push and $\lg n + O(1)$ comparisons for pop" [19] was presented at the 10th International Computer Science Symposium in Russia held in Listvyanka in July 2015.

✉ Jyrki Katajainen
jyrki@di.ku.dk

Stefan Edelkamp
edelkamp@tzi.de

Amr Elmasry
elmasry@alexu.edu.eg

[1] Faculty 3—Mathematics and Computer Science, University of Bremen, PO Box 330 440, 28334, Bremen, Germany

[2] Department of Computer Engineering and Systems, Alexandria University, Alexandria, 21544, Egypt

[3] Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100, Copenhagen East, Denmark

Keywords In-place data structures · Priority queues · Binary heaps · Comparison complexity · Constant factors

1 Introduction

In its elementary form, a priority queue is a data structure that stores a multiset of elements and supports the operations: *construct*, *minimum*, *insert*, and *extract-min* [11, Chapter 6]. In most cases, in applications where this set of operations is sufficient, the basic priority queue that the users would select is a binary heap [49]. Even though a binary heap is practically efficient, its theoretical behaviour is known not to be optimal. The same is true for the best-known variants presented in the literature (see, e.g. [27]).

In the theory of *in-place* data structures, an infinite array is assumed to be available and, at any given point of time, the elements currently manipulated must be kept in the first positions of this array. Thus, with respect to the dynamization of the array, the issues related to memory management are abstracted away. Throughout the paper we use n to denote the number of elements stored in the data structure prior to the operation in question. In addition to the array of elements, a constant amount of space is assumed to be available for storing elements and variables of $O(\lg n)$ bits. We assume the comparison-based model of computation, where the elements can be constructed, moved, compared, overwritten, and destroyed, but it is not allowed to modify the elements in any other way. When measuring the running time, these operations on elements and the normal arithmetic, logical, and bitwise operations on variables are assumed to have a unit cost each.

In this paper, we present a variant of a binary heap, named strengthened lazy heap, that

(1) can store any multiset of elements (duplicates allowed);
(2) operates in-place so that it uses $O(1)$ extra space in addition to the elements maintained at the beginning of an array;
(3) supports *minimum* in $O(1)$ worst-case time with no element comparisons;
(4) supports *insert* in $O(1)$ worst-case time;
(5) supports *extract-min* in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons.[1]

Assume that, for a problem of size n, the bound achieved for a consumed resource is $\mathrm{A}(n)$ and the best possible bound is $\mathrm{OPT}(n)$. We distinguish three different concepts of optimality:

Asymptotic optimality: $\mathrm{A}(n) = O(\mathrm{OPT}(n))$.
Constant-factor optimality: $\mathrm{A}(n) = \mathrm{OPT}(n) + o(\mathrm{OPT}(n))$.
Up-to-additive-constant optimality: $\mathrm{A}(n) = \mathrm{OPT}(n) + O(1)$.

As to the running times, the bounds we achieved are asymptotically optimal. As to the number of element comparisons performed, the bounds are optimal to within additive

[1] As is standard, throughout the text, for integers $d \geq 2$ and $n \geq 0$, we write $\log_d n$ when we mean $\log_d (\max\{d, n\})$, and we use $\lg n$ as a shorthand for $\log_2 n$.

constant terms. The claim follows from the information-theoretic lower bound for sorting [35, Section 5.3.1]. Namely, if a priority queue is used for sorting and every *insert* requires $O(1)$ worst-case time, some *extract-min* must perform at least $\lg n - O(1)$ element comparisons. Note that if we assume integer elements and allow for word-RAM computations on them to have unit cost each, it is possible to break the above bounds [28, 29, 43].

For a binary heap, the number of element moves performed by *extract-min* is at most $\lg n + O(1)$. We have to avow that, for our data structure, *extract-min* requires more element moves. On the positive side, we can adjust the number of element moves to be at most $\lg n + O(\lg^{(\gamma)} n)$ (the logarithm taken γ times), for any fixed integer $\gamma \geq 1$, while still achieving the same bounds for the other operations. Another minor drawback for our data structure is that the number of element comparisons per *construct* is increased from $(13/8)n + o(n)$, the best bound known for a binary heap [10], to $(23/12)n + o(n)$.

When strengthened lazy heaps are used in different applications, better comparison bounds can be derived for the underlying tasks. We only give two examples. First, when a strengthened lazy heap is used in heapsort [49], the resulting algorithm sorts n elements in-place in $O(n \lg n)$ worst-case time involving at most $n \lg n + O(n)$ element comparisons. The number of element comparisons performed matches the information-theoretic lower bound for sorting up to the additive linear term. Ultimate heapsort [33] is known to have the same complexity bounds, but in both solutions the constant factor of the additive linear term is high. Note that several other variants of heapsort (e.g. those discussed in [24, 25, 47, 48, 50]) are not constant-factor optimal with respect to the number of element comparisons performed, or the additive term may be asymptotically higher (e.g. for those discussed in [6, 27, 45, 51, 52]). Second, when a strengthened lazy heap is used in adaptive heapsort [36], the resulting algorithm sorts a sequence X of length n having $Inv(X)$ inversions in asymptotically optimal worst-case time $O(n \lg(Inv(X)/n))$ performing at most $n \lg(Inv(X)/n) + O(n)$ element comparisons, which is optimal to within the additive linear term. This solution is not fully in-place, but it would use less memory than the solution relying on a weak heap [15].

Ever since the work of Williams [49], it was open whether there exists an in-place priority queue that can match the information-theoretic lower bounds on the number of element comparisons for all the operations. In view of the lower bounds proved in [27], it was not entirely clear if such a structure at all exists. In this paper we answer the question affirmatively by introducing the strengthened lazy heap that operates in-place, supports *minimum* and *insert* in $O(1)$ worst-case time, and *extract-min* in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons.

Although, compared to a binary heap, we surpass the comparison bound for *extract-min* and the time bound for *insert* [27], our data structure is similar to a binary heap. As this result suggests, the following two alterations are crucial:

(1) To improve the comparison bound for *extract-min*, we reinforce a stronger heap order at the bottom levels of the heap such that the element at any right child is not smaller than that at its left sibling.

(2) To speed up *insert*, we buffer insertions and allow a poly-logarithmic number of nodes to violate heap order in relation to their parents.

2 Historical Notes and Earlier Attempts

We kick off by reviewing the fascinating history of the problem of optimizing priority queues with respect to the extra space used and the number of element comparisons performed by the basic operations.

A *binary heap* (see Fig. 1) is an in-place priority queue that is viewed as a nearly-complete binary tree [11, Section B.5.3], the nodes of which are stored in an array in breadth-first order. For every node other than the root, the element at that node is not smaller than the element at its parent (*heap order*). A binary heap supports *minimum* in $O(1)$ worst-case time, and *insert* and *extract-min* in $O(\lg n)$ worst-case time, n being the number of current elements. For Williams' original proposal [49], in the worst case, the number of element comparisons performed by *insert* is at most $\lceil \lg n \rceil$ and that by *extract-min* at most $2 \lg n$. Immediately after the appearance of Williams' paper, Floyd showed [26] how to support *construct*, which builds a heap for n elements, in $O(n)$ worst-case time with at most $2n$ element comparisons.

The operations used to reestablish heap order after an insertion or an extraction come with different names; in the literature people use the terms: bubbling, sifting, or heapifying. To fix the terminology, we give a complete description of the binary-heap class in pseudo-code in Fig. 2. In *insert*, after augmenting the first vacant array entry as the last leaf, *sift-up* traverses the path from that leaf to the root and moves the encountered elements one level down until the correct place of the new element is found, where the newcomer is inserted and all the elements on the traversed path are in sorted order. In *extract-min*, after temporarily keeping the last element of the array out, *sift-down* traverses the so-called *special path* starting from the root and going at each level to the child that holds the smaller of the elements stored at the two siblings, moves the encountered elements one level up until the correct place of the element we kept out is found, where this element is added back and the elements on the special path are in sorted order. The main problem is that, in their original form [49], *sift-up* requires at most $\lceil \lg n \rceil$ element comparisons and *sift-down* at most $2 \lg n$ element comparisons. In accordance, we shall introduce other variants for implementing and using these procedures.

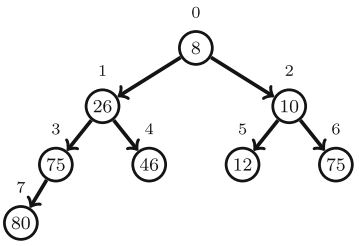

Fig. 1 A binary heap in an array $\mathbf{a}[0\colon 8) = [8, 26, 10, 75, 46, 12, 75, 80]$ viewed as a nearly-complete binary tree. The array indices are written beside the nodes

```
class binary-heap

private

data n: index, a: element[ ]

procedure left-child
input i: index
output index
return 2i + 1

procedure right-child
input i: index
output index
return 2i + 2

procedure parent
input i: index
output index
if i = 0:
  | return 0
return ⌊(i − 1)/2⌋

procedure sift-down
input i: index, n: index
x ← a[i]
while left-child(i) < n:
  | j ← left-child(i)
  | if right-child(i) < n:
  |   | if a[right-child(i)] < a[j]:
  |   |   | j ← right-child(i)
  | if not (a[j] < x):
  |   | break
  | a[i] ← a[j]
  | i ← j
a[i] ← x

procedure sift-up
input j: index
x ← a[j]
while j ≠ 0:
  | i ← parent(j)
  | if not (x < a[i]):
  |   | break
  | a[j] ← a[i]
  | j ← i
a[j] ← x
```

```
public

procedure construct
input m: index, b: element[ ] as reference
n ← m
a ← b
if n < 2:
  | return
for i ← parent(n − 1) down to 0:
  | sift-down(i, n)

procedure minimum
output element
assert n > 0
return a[0]

procedure insert
input x: element
a[n] ← x
sift-up(n)
n ← n + 1

procedure extract-min
assert n > 0
n ← n − 1
if n ≠ 0:
  | a[0] ← a[n]
  | sift-down(0, n)
```

Fig. 2 A complete description of a binary-heap class in pseudo-code; the private part on the left and the public part on the right. Normally, parameters are passed to a procedure by value, but in *construct* the element array **b** is passed by reference. Correspondingly, in the assignment **a** ← **b** the address of **b** is copied, not the elements

A binary heap is a simple data structure. Both the *sift-up* and *sift-down* execute a single step of insertionsort to insert a new element into a sorted path and keep the elements on the path sorted. In practice, when problems to be solved fit into main memory, a binary heap is also efficient. When the operation sequence only consists of insertions, *insert* runs in $O(1)$ time on the average [31], i.e. under the assumption that the given elements are uniformly distributed. Also, by modifying

sift-down so that it first follows the special path down to a leaf and then puts the sifted element in its place on the way up, as proposed in [35, Exercise 5.2.3–18], *extract-min* performs $\lg n + O(1)$ element comparisons on the average [2, 41], assuming that the operation sequence only consists of minimum extractions. As a consequence, in software libraries, an elementary priority queue is often implemented as a binary heap. This is in contrast to the data structures to be presented in this paper; we aim at improving the theoretical worst-case performance but make no claims about the practical applicability of the solutions presented.

As a binary heap does not support all the operations optimally, many attempts have been made to develop priority queues supporting the same set (or even a larger set) of operations that improve the worst-case running time of the operations as well as the number of element comparisons performed by them [4, 7, 9, 10, 17, 20, 22, 27, 32, 44]. In Table 1 we summarize the history of the problem, focusing on the best-known space and comparison complexities.

A d-ary heap, proposed by Johnson [32], is a natural extension of a binary heap, where every branch node has $d \geq 2$ children, except possibly the last branch node. The pseudo-code for a d-ary heap is similar to that shown in Fig. 2. However, the formulas to compute the children and parent of a node are different, *construct* has the arity as an additional parameter, and in *sift-down* at each branch one has to find the smallest of the elements stored at its children. This minimum finding requires $d-1$ element comparisons and one more comparison is needed to determine whether or not we can stop the traversal. Since the height of a d-ary heap of size n is $\mathbf{h}(n,d) \stackrel{\text{def}}{=} \lceil \log_d(dn-n+1) \rceil$ (the height of a single node being 1), *insert* involves at most $\mathbf{h}(n+1,d)-1$ element comparisons and *extract-min* at most $d \times (\mathbf{h}(n,d)-1)$ element comparisons. There is an interesting trade-off between the cost of these two operations: By making d larger, *insert* becomes faster but *extract-min* becomes slower. Setting $d = \lfloor n^{1/k} \rfloor$, the worst-case running time of *insert* becomes $O(k)$.

For binary heaps, Gonnet and Munro showed [27] how to perform *insert* using at most $\lg \lg n + O(1)$ element comparisons and *extract-min* using at most $\lg n + \log^* n + O(1)$ element comparisons.[2] Chen et al. showed [10] how to construct a binary heap in-place using at most $(13/8)n + o(n)$ element comparisons in $O(n)$ worst-case time. Carlsson et al. showed [9] how to achieve $O(1)$ worst-case time per *insert* by an in-place data structure that utilizes a queue of pennants. (A *pennant* is a binary heap with an extra root that has one child.) For this data structure, the number of element comparisons performed per *extract-min* is bounded by $3 \lg n + \log^* n + O(1)$.

The binomial queue [44] was the first priority queue supporting *insert* in $O(1)$ worst-case time and *extract-min* in logarithmic worst-case time. (This was mentioned as a short note at the end of Brown's paper [4].) However, the binomial queue is a pointer-based data structure requiring $O(n)$ pointers in addition to the elements. The multipartite priority queue [20] was the first priority queue achieving the asymptotically optimal time and up-to-additive-constant-optimal

[2] $\log^* n$ is the iterated logarithm of n, recursively defined as 1 for $n \leq 2$, and $1 + \log^*(\lg n)$ for $n > 2$.

Table 1 The performance of some elementary priority queues—the data structures in the top and bottom sections are in-place and those in the middle section require linear extra space. The complexity of operations is expressed in the number of element comparisons performed in the worst case, unless otherwise stated; n denotes the number of elements stored in the data structure prior to the operation in question. For all these data structures, the worst-case running time of *minimum* is $O(1)$ with no element comparisons and that of *construct*, *insert*, and *extract-min* is proportional to the number of element comparisons performed except for heaps on heaps

Data structure	*construct*	*insert*	*extract-min*
binary heaps [26, 49]	$2n$	$\lceil \lg n \rceil$	$2 \lg n$
d-ary heaps [32, 42]	$(d/(d-1))n$	$\log_d n + O(1)$	$d \log_d n + O(1)$
heaps on heaps [27]	$2n$	$\lg \lg n + O(1)^{\ddagger}$	$\lg n + \log^* n + O(1)$
in-place heaps on heaps [10]	$(13/8)n + o(n)$	$\lg \lg n + O(1)^{\ddagger}$	$\lg n + \log^* n + O(1)$
queues of pennants [9]	$O(n)$	$O(1)$	$3 \lg n + \log^* n + O(1)$
binomial queues [4, 44]	$n - 1$	2	$2 \lg n + O(1)$
multipartite priority queues [20]	$O(n)$	$O(1)$	$\lg n + O(1)$
engineered weak heaps [17]	$n - 1$	$O(1)$	$\lg n + O(1)$
penultimate binary heaps [22]	$(13/8)n + o(n)$	amortized $O(1)$	amortized $\lg n + O(1)$
strong heaps [Section 4]	$(23/12)n + o(n)$	$2 \lg n$	$3 \lg n$
lazy heaps [Section 5]	$(13/8)n + o(n)$	amortized $8 + o(1)$	$\lg n + \log^* n + O(1)$
relaxed lazy heaps [Section 6]	$(13/8)n + o(n)$	$O(1)$	$\lg n + \log^* n + O(1)$
strengthened lazy heaps [Section 7]	$(23/12)n + o(n)$	$O(1)$	$\lg n + O(1)$

‡The running time of *Insert* is logarithmic

comparison bounds. Unfortunately, the structure is involved and its representation requires $O(n)$ pointers. Another solution [17] is based on weak heaps [14]: To implement *insert* in $O(1)$ worst-case time, a bulk-insertion strategy is used—employing two buffers and incrementally merging one with the weak heap before the other is full. The weak heap also achieves the desired worst-case time and comparison bounds, but it uses n additional bits.

In [22], another in-place variant of binary heaps was introduced that supports *insert* in $O(1)$ time and *extract-min* in $O(\lg n)$ time with at most $\lg n + O(1)$ element comparisons, all in the amortized sense. The components used are a partial heap, an insertion buffer, and a backlog. The invariant is that none of the elements in the backlog is smaller than the elements in the buffer and main heap. This allows *extract-min* to borrow an element from the backlog that is known to go down all the way to the bottom-most level of the heap, so that no additional bottom-up search is needed. Once the backlog is empty, the structure is split again; a costly operation that amortizes over time. (The backlog idea was inspired from ultimate heapsort [33], which computes the median in linear time to partition the data into two parts.) The insertion buffer is used to accumulate elements for bulk insertions. The size of the buffer is bounded above by about $\lg^2 n$ to allow efficient bulk insertion into a heap. Furthermore, by deamortization, *insert* could be supported in $O(1)$ worst-case time,

and simultaneously *construct* and *extract-min* could be supported as efficiently as for binary heaps. Finally, using another deamortization argument, it was shown that a sequence of *extract-min* operations—with no intermixed *insert* operations—can be performed in-place using worst-case $\lg n + O(1)$ element comparisons per operation (after linear preprocessing).

Our work shows the limitation of the lower bounds proved by Gonnet and Munro [27] (see also [7]) in their prominent paper on binary heaps. Gonnet and Munro showed that $\lceil \lg \lg(n+2) \rceil - 2$ element comparisons are necessary to insert an element into a binary heap. In addition, slightly correcting [27], Carlsson [7] showed that $\lceil \lg n \rceil + \delta(n)$ element comparisons are necessary to extract the minimum from a binary heap that has n elements, where $\delta(n)$ is the largest integer satisfying $h_{\delta(n)} \leq \lg n - 2$ and $h_1 = 1$, $h_i = h_{i-1} + 2^{h_{i-1}+i-1}$. One should notice that these lower bounds are only valid under the following assumptions:

(1) All the elements are stored in one nearly-complete binary tree.
(2) Every node obeys the heap order before and after each operation.
(3) No order relation among the elements of the same level can be deduced from the element comparisons performed by previous operations.

In this paper, we bypass those lower bounds: We prove that the number of element comparisons for *extract-min* can be lowered to at most $\lg n + O(1)$ if we overrule the third assumption by imposing an additional requirement that the element at any right child is not smaller than that at the left sibling. We also prove that *insert* can be performed in $O(1)$ worst-case time if we overrule the second assumption by allowing poly-logarithmic number of nodes to violate heap order. We then combine the two ideas in our final data structure.

3 Intuition and Techniques

Our starting point is a binary heap with n elements stored at the beginning of an array. Because of the lower bounds known for *insert* and *extract-min* [27], we cannot expect the elements to be maintained in heap order if we want to surpass these lower bounds. In the literature, the following approaches have been used to improve *insert* and *extract-min*:

Relaxing heap order: In a relaxed heap [13, 15], some elements are allowed to violate heap order such that an element at a node may be smaller than that at its parent. Although the number of potential violation nodes are restricted to be logarithmic, a separate data structure is needed to keep track of those violations.

Weakening heap order: In a weak heap [14, 15], the elements are half-ordered so that none of the elements in the right subtree of a node is smaller than the element at that node. Additionally, the root has no left subtree so the minimum is stored at the root. The elements are still stored in an array, but for each node one additional *reverse* bit is maintained that tells which of its two children roots the right subtree. The virtue of the structure is that the two subtrees of a node can be swapped by flipping the reverse bit.

Strengthening heap order: In a fine heap [8] (see also [37, 46]), in addition to the regular heap order, an extra *dominance* bit is stored at each node to indicate which of its two children contains the smaller element.

For a relaxed heap, *insert* could be supported in $O(1)$ worst-case time by attaching the inserted element into the heap, marking it as potentially violating, and applying some local transformations to reduce the number of potential violation nodes when there are too many of them. For a weak heap and a fine heap, *extract-min* can be supported with $\lg n + O(1)$ element comparisons since it is possible to navigate down the heap along the special path without performing any element comparisons by using the bits stored at the nodes.

We can draw the conclusion that these approaches may be useful for our purposes, even if the alternatives of having less order or having more order are conflicting. In our final construction, to improve *insert*, we allow $O(\lg^2 n)$ nodes to violate heap order, and to improve *extract-min*, we maintain $O(n/\lg n)$ nodes at the top levels in heap order and the remaining bottom nodes in a stronger heap order. What makes the matter delicate is that we only have $O(1)$ extra words of space available to keep track of where the different types of nodes—violating, normal, and strengthened—are.

To develop our data structures, we had to use a variety of techniques listed below. These techniques are not new, but the combination of how we use them is. The standard reference, where many of the techniques are discussed, is the book by Overmars [39]. Our other inspirational source was the stratified trees developed by van Emde Boas et al. (for a historical survey, see [23]).

Binary search and stopovers: In a heap-ordered tree, the elements along any path from a leaf to the root appear in sorted order. Hence, when applying *sift-up* or *sift-down* the correct position of a new element can be found by binary search. This observation leads to *insert* which involves at most $\lg \lg n + O(1)$ element comparisons [6, 27]. In *extract-min*, *sift-down* can be refined by making several stopovers on the way down and applying binary search for a subpath: Go first $\lg n - \lg \lg n$ levels down along the special path and test whether the searched position is above or below this point. If it is above, complete the search by performing binary search between the current point and the previous stopover. Otherwise, apply the same procedure recursively below. This stopover optimization leads to *extract-min* which involves at most $\lg n + \log^* n + O(1)$ element comparisons [7, 27].

Stratification: In a stratified tree [23], each leaf of the top tree roots a bottom tree. Even though this decomposition can be used recursively, we only use these two layers. For our structure, in the top tree the normal heap order is maintained, whereas in each of the bottom trees the heap order is strengthened. For binary heaps, this idea was earlier used for speeding up in-place heap construction [10].

Buffering and partial rebuilding: Buffering is a well-known technique that can be used to improve *insert* in priority queues [1]. The idea is to insert the given elements into a buffer that is coupled back to back with the heap and, at some point, combine the buffer with the heap and partially rebuild the structure. In the literature, people use the terms: bulk insertions, batch insertions, or group insertions. In the standard construction [1], extra space is needed in the form of pointers. To combine the buffer with the heap, we modify Floyd's heap-construction algorithm

[26] such that, instead of constructing a whole heap, we use the same strategy to add a bulk of size $O(\lg^2 n)$ or $O(\lg^3 n)$ to the existing heap.

Global rebuilding: As for a weight-balanced tree [38], it might be a good idea to let a data structure go out of balance until the operations become inefficient and then make the structure perfectly balanced. We need this kind of global rebuilding when maintaining the border between the top tree and the bottom trees. The only requirement set for the bottom trees is that their height may deviate from $\lg \lg n$ by only some constant in either direction. Global rebuilding will be used to ensure that this is the case at all times while the value of n is changing.

Deamortization: If partial rebuilding or global rebuilding was done in one go, the worst-case running time of a single operation might become high. Hence, rebuilding is done incrementally by distributing the work evenly among a set of operations so that the cost of each operation sharing the work only increases by a constant. Here, it is significant that the state of an incremental process can be recorded using a constant amount of space.

In [17], we discussed how a related incremental process—deamortization of bulk insertions to get constant worst-case-time *insert* in a weak heap—could be implemented (for the source code, see [16]). The computation was described as a state machine and, in connection with each modifying operation, a constant number of state transitions were made. This kind of deamortization is customarily considered trivial in the algorithm literature, but seldom seen programmed. According to the benchmark results reported in [17], a heap-construction algorithm based on repeated insertions using deamortized constant-time *insert* was an order of magnitude slower than a specialized heap-construction algorithm.

Heaps in heaps: Many priority queues are organized as a collection of heaps (see, e.g. [5, 15, 21, 30, 40, 44]). In our in-place setting, a heap is maintained in an array, but inside this array, in separate subarrays, up to three multiary heaps are maintained in order to keep track of the elements that are still out of order with respect to the elements in the main heap.

Mirroring: In the front-to-back view, when a heap of size n is organized in an array $\mathbf{a}[0\colon n)$, $\mathbf{a}[0]$ stores the element at the root and the tree grows to the right. When space is tight, we also have to rely on the back-to-front view, where $\mathbf{a}[n-1]$ keeps the element at the root and the tree grows to the left. For such a *mirrored heap*, the operations are programmed as before, except that all indices are negative with respect to the base address $\mathbf{a}+n-1$.

Concurrency management: In our final construction, there are three incremental processes that may be ongoing when an insertion or an extraction is executed. Our main goal is to ensure that there is no undesirable interference between the ongoing processes and the priority-queue operations being executed. This turned out to be the most complicated part of our construction because of the lack of the convenient programming-language mechanisms to describe the underlying processes and their interactions.

We develop and present our data structures in sequence in the form of basic building blocks: (1) strong heaps (Section 4) that maintain a stronger heap order, but

are slow, and (2) lazy heaps (Section 5) that delay insertions by buffering and perform occasional bulk insertions instead, leading to amortized $O(1)$ time per *insert*. Thereafter, we show how to carry out bulk insertions incrementally in a deamortized solution (Section 6) supporting *insert* in $O(1)$ worst-case time. For that, the boundary of the buffer is stored as a couple of heaps in the main heap, one of them is mirrored. Relying on buffering, stratification of a binary heap on top of many small strong heaps, maintaining the border by global rebuilding, and employing binary search while sifting, we end up with the best results (Section 7): $O(1)$ worst-case time per *insert* and $O(\lg n)$ worst-case time with at most $\lg n + O(1)$ element comparisons per *extract-min*. This final structure is complicated and requires concurrency management to avoid the interference of several incremental processes.

Also, to efficiently support the *construct* operation in-place, we use the following standard in-place techniques.

Permuting in-place: Assume that we are given an array of elements $\mathbf{a}[0\colon t)$ and a permutation $\sigma(0)\sigma(1)\cdots\sigma(t-1)$. It is well known [34, Section 1.3.3] that the task of permuting the elements in this array in-place can be accomplished in $O(t)$ worst-case time (using $O(t)$ words of memory) by following the cycle structure of the permutation. In the worst case, the number of element moves performed would be t plus the number of cycles in the given permutation, for a total of at most $\lfloor (3/2)t \rfloor$ element moves.

Packing small integers: Consider permuting the elements of a small subheap of size t, the root of which is at some specific index. To optimize the number of element moves performed by our procedures, we could use an array of pointers that refer to the elements, and then manipulate these pointers instead of moving the elements themselves (cf. address-table sorting [35, Section 5.2]). More specifically, each entry of this array could store a pair of small integers: one giving the level of an element relative to that of the root, and another giving the offset at that level. For $t \stackrel{\text{def}}{=} O(\lg n / \lg\lg n)$, the size of each small pointer is $O(\lg\lg n)$ bits, and the whole representation of this subheap could be encoded in $O(\lg n)$ bits. According to our assumption on the word size, these integers can be packed into a constant number of variables. Using the normal arithmetic, logical, and bitwise operations, each integer can be read and written in $O(1)$ worst-case time.

4 Strong Heaps: Adding More Order

A *strong heap* is a binary heap with one additional invariant: The element at any right child is not smaller than that at the left sibling. This left-dominance property is fulfilled for every right child in a fine heap [8] (or its alternatives [37, 46]), which uses one extra bit per node to maintain the property. On the contrary, a strong heap operates in-place, but its operations are slower. Like a binary heap, a strong heap is viewed as a nearly-complete binary tree where the lowest level may be missing some nodes at the rightmost positions. In addition, this tree is embedded in an array in the same way, i.e. the formulas for computing *left-child*, *right-child*, and *parent* are still the same (see Fig. 2).

Two views of a strong heap are exemplified in Fig. 3. On the left, the directed acyclic graph has a nearly-complete binary tree as its *skeleton*: There are arcs from every parent to its children and additional arcs from every left child to its sibling indicating the dominance relations. On the right, in the *stretched tree*, the arcs from each parent to its right child are removed as these dominance relations can be induced. In the stretched tree a node can have 0, 1, or 2 children. A node has one child if in the skeleton it is a right child that is not a leaf or a leaf that has a right sibling. A node has two children if in the skeleton it is a left child that is not a leaf. If the skeleton has height h (height of a single node being 1), the height of the stretched tree is at most $2h - 1$, and on any root-to-leaf path in the stretched tree the number of nodes with two children is at most $h - 2$.

When implementing *construct* and *extract-min* for a binary heap, the basic primitive used is the *sift-down* procedure [26, 49]. For a strong heap, the *strengthening-sift-down* procedure has the same purpose, and our implementation (see Fig. 4) is similar, with one crucial exception that we operate with the stretched tree instead of the nearly-complete binary tree. As for a binary heap, *extract-min* can be implemented (also in Fig. 4) by replacing the element at the root with the element at the last position of the array (if there is any) and then invoking *strengthening-sift-down* for the root.

Example 1 Consider the strong heap in Fig. 3. If its minimum was replaced with the element 17 taken from the end of the array, the path to be followed by *strengthening-sift-down* would include the nodes $\langle 3, 4, 5, 7, 11 \rangle$.

Let n denote the size of a strong heap and h the height of the underlying tree skeleton. When going down the stretched tree, we have to perform at most $h - 2$ element comparisons due to branching at binary nodes and at most $2h - 1$ element

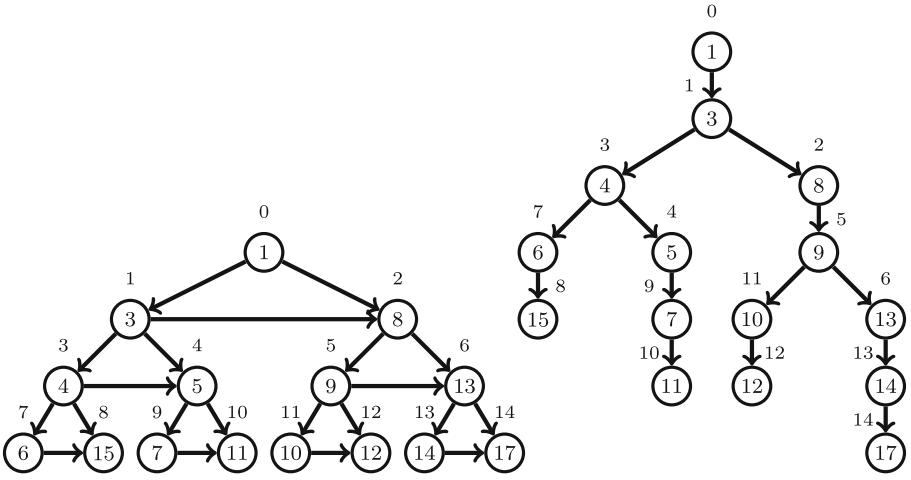

Fig. 3 A strong heap in an array $\mathbf{a}[0\colon 15) = [1, 3, 8, 4, 5, 9, 13, 6, 15, 7, 11, 10, 12, 14, 17]$ viewed as a directed acyclic graph (left) and a stretched tree (right)

```
class strong-heap

private

data n: index, a: element[]

procedure even
input i: index
output Boolean
return (i mod 2) = 0

procedure sibling
input i: index
output index
if i = 0:
  return 0
return i + even(i + 1) − even(i)

procedure is-leaf
input i: index, n: index
output Boolean
if not even(i):
  return sibling(i) ≥ n
return left-child(i) ≥ n

procedure strengthening-sift-down
input i: index, n: index
x ← a[i]
while not is-leaf(i, n):
  j ← sibling(i)
  if even(i):
    j ← left-child(i)
  else if j < n and left-child(i) < n and
        not (a[j] < a[left-child(i)]):
    j ← left-child(i)
  if not (a[j] < x):
    break
  a[i] ← a[j]
  i ← j
a[i] ← x

public

procedure construct
input m: index, b: element[] as reference
n ← m
a ← b
if n < 2:
  return
for i ← n − 1 down to 0:
  strengthening-sift-down(i, n)

procedure extract-min
assert n > 0
n ← n − 1
if n ≠ 0:
  a[0] ← a[n]
  strengthening-sift-down(0, n)
```

Fig. 4 Implementation of *strengthening-sift-down*, and its use in *construct* and *extract-min* in a strong heap organized in an array $\mathbf{a}[0\colon n)$; a right child is never accessed directly

comparisons due to checking whether to stop or not. Hence, the number of element comparisons performed by *extract-min* is bounded by $3h - 3$, which is at most $3 \lg n$ as $h = \lfloor \lg n \rfloor + 1$.

To build a strong heap, we mimic Floyd's heap-construction algorithm [26]; that is, we invoke *strengthening-sift-down* for all nodes, one by one, processing them in reverse order of their array positions (see Fig. 4). One element comparison is needed for every met left child in order to compare the element at its right sibling with that at its left child, making a total of at most $n/2$ element comparisons. The number of other element comparisons is bounded by the sum $\sum_{i=1}^{\lfloor \lg n \rfloor + 1} 3 \cdot i \cdot \lceil n/2^{i+1} \rceil$, which is at most $3n + o(n)$. Thus, *construct* requires at most $3.5n + o(n)$ element comparisons.

For *construct* and *extract-min*, the amount of work done is proportional to the number of element comparisons performed, i.e. the worst-case running time of *construct* is $O(n)$ and that of *extract-min* is $O(\lg n)$.

Lemma 1 *A strong heap of size n can be built in $O(n)$ worst-case time by repeatedly calling strengthening-sift-down. Each strengthening-sift-down uses $O(\lg n)$ worst-case time and performs at most $3 \lg n$ element comparisons.*

Analogously with *strengthening-sift-down*, it would be possible to implement *strengthening-sift-up*. As the outcome, *insert* would perform at most $2 \lg n$ element comparisons.

Next we show how to perform a *sift-down* operation on a strong heap of size n with at most $\lg n + O(1)$ element comparisons. At this stage we allow the amount of work to be higher, namely $O(n)$. To achieve the better comparison bound, we have to assume that the heap is *complete*, i.e. that all leaves have the same depth. Consider the case where the element at the root of a strong heap is replaced by a new element. In order to reestablish strong heap order, the *swapping-sift-down* procedure (Fig. 5) traverses the left spine of the skeleton bottom up starting from the leftmost leaf, and determines the correct place of the new element, using one element comparison at each node visited. Thereafter, it moves all the elements above this position on the left spine one level up, and inserts the new element into its correct place. If this place is at height g, we have performed g element comparisons. Up along the left spine there are $\lg n - g + O(1)$ remaining levels to which we have moved other elements. While this results in a heap, we still have to reinforce the left-dominance property at these upper levels. In accordance, we compare each element that has moved up with the element at the right sibling. If the element at index j is larger than the element at index $j+1$, we interchange the subtrees T_j and T_{j+1} rooted at positions j and $j+1$ by swapping all their elements. The procedure continues this way until the root is reached.

```
procedure swap
input/output x: element as reference,
             y: element as reference
z ← x
x ← y
y ← z

procedure swap-subtrees
input u: index, v: index, n: index
j ← 1
while v < n:
    for i ← 0 to j − 1:
        swap(a[u + i], a[v + i])
    u ← left-child(u)
    v ← left-child(v)
    j ← 2 ∗ j

procedure leftmost-leaf
input i: index, n: index
output index
while left-child(i) < n:
    i ← left-child(i)
return i

procedure bottom-up-search
input i: index, j: index
output index
while i < j and not (a[i] < a[j]):
    j ← parent(j)
return j

procedure lift-up
input i: index, j: index, n: index
x ← a[j]
a[j] ← a[i]
while i < j:
    swap(a[parent(j)], x)
    if a[sibling(j)] < a[j]:
        swap-subtrees(j, sibling(j), n)
    j ← parent(j)

procedure swapping-sift-down
input i: index, n: index
k ← leftmost-leaf(i, n)
k ← bottom-up-search(i, k)
lift-up(i, k, n)
```

Fig. 5 Implementation of *swapping-sift-down* for a complete strong heap

Example 2 Consider the strong heap in Fig. 3. If the element at the root was replaced with the element 16, the left spine to be followed by *swapping-sift-down* would include the nodes $\langle (3), (4), (6) \rangle$, the new element would be placed at the last leaf we ended up with, the elements on the left spine would be lifted up one level, and an interchange would be necessary for the subtrees rooted at node (6) and its new sibling (5).

Given two complete subtrees of height h, the number of element moves needed to interchange the subtrees is $O(2^h)$. As $\sum_{h=1}^{\lfloor \lg n \rfloor} O(2^h)$ is $O(n)$, the total work done in the subtree interchanges is $O(n)$. Thus, *swapping-sift-down* requires at most $\lg n + O(1)$ element comparisons and $O(n)$ work.

Lemma 2 *In a complete strong heap of size n, swapping-sift-down runs in-place and uses at most* $\lg n + O(1)$ *element comparisons and* $O(n)$ *moves.*

To improve *construct* with respect to the number of element comparisons and the number of element moves performed, still keeping the worst-case running time linear, we can use the algorithms of Carlsson et al. [8] developed for a fine heap. Instead of *swapping-sift-down*, the subtree interchanges are realized by flipping the dominance bits. The basic algorithm [8, Lemma 3.2] builds a fine heap of size n in $O(n)$ worst case time with at most $2n + o(n)$ element comparisons. An interesting change in the base case [8, Lemma 3.3] leads to an improvement reducing the number of element comparisons performed to $(23/12)n + o(n)$. In accordance, for a strong heap, we would expect an in-place construction algorithm with at most $(23/12)n + o(n)$ element comparisons. For a strong heap we want the elements to be placed at their correct final locations, while for a fine heap it is the case that guiding information encoded in bits is sufficient. To this extent, at the end, we have to transform a fine heap with bits to a strong heap.

The algorithm set-up is as follows. As advised in [10], we divide the tree into a top tree and a set of bottom trees such that each leaf of the top tree roots a small bottom tree (stratification). To optimize the number of element moves done by the construction, we fix the level where we cut the tree such that the size of each bottom tree is less than, but as close as possible to, $t \stackrel{\text{def}}{=} \lg n / \lg \lg n$. The algorithm makes each of the bottom trees a strong heap, one after the other, and then processes the nodes at the top tree. The key observation is that most of the work is done when creating the bottom heaps, so that for the nodes at the top tree any sifting strategy can be used while only affecting the low-order terms in the computational complexity. We thus end up applying *strengthening-sift-down* on the $o(n)$ nodes of the top tree.

One bottom tree may be nearly complete, not complete, so we process it separately by using *strengthening-sift-down*. For each of the remaining complete bottom trees, B_j, we do the following.

(1) Create an array of small pointers referring to the elements of B_j and reserve an array of bits for the dominance relations between the siblings.

(2) Use the algorithm of Carlsson et al. [8, Lemma 3.3] to build a fine heap for the elements in B_j. Here no element moves are done; all subtree interchanges are emulated by flipping the dominance bits and all element moves are emulated by swapping small pointers.
(3) Perform a preorder traversal of the resulting fine heap to create a permutation that indicates the destination of each element. Again small pointers are used when specifying the destinations. Similarly, the recursion stack maintained can just record small pointers.
(4) Permute the elements to get them to their final destinations (permuting in-place).

By packing the pointers used, the construction of each bottom heap only requires $O(t \times \lg t)$ bits, which is $O(\lg n)$ bits or $O(1)$ variables (packing small integers). When one bottom heap is constructed, the space occupied can be freed and reused for the construction of another bottom heap. Each of the above steps requires linear time, so the overall running time is linear too.

The results presented in this section can be summarized as follows.

Theorem 1 *Let n denote the number of elements stored in a data structure. A strong heap is an in-place priority queue, for which*

(1) *construct requires $O(n)$ worst-case time performing at most $(23/12)n + o(n)$ element comparisons;*
(2) *minimum requires $O(1)$ worst-case time involving no element comparisons;*
(3) *insert requires $O(\lg n)$ worst-case time involving at most $2 \lg n$ element comparisons; and*
(4) *extract-min requires $O(\lg n)$ worst-case time involving at most $3 \lg n$ element comparisons. Furthermore, given a mechanism to keep the strong heap complete, extract-min can be refined to perform $\lg n + O(1)$ element comparisons, but this would increase its worst-case running time to $O(n)$.*

5 Lazy Heaps: Buffering Insertions

In the variant of a binary heap that we describe in this section some nodes may violate heap order because insertions are buffered and only partially ordered bulks are inserted into the heap. The main difference between the present construction and the construction in [17] is that, for a data structure of size n and a fixed integer $\alpha \geq 2$, we allow $O(\lg^\alpha n)$ heap-order violations instead of $O(\lg n)$, but we still use $O(1)$ extra space to track where the potential violations are. Using *strengthening-sift-down* instead of *sift-down*, the construction will also work for a strong heap.

A *lazy heap* is composed of two parts, *main heap* and *insertion buffer*, that are laid out back to back in an array; the insertion buffer comes after the main heap. The following rules are imposed:

(1) *insert* adds every new element to the insertion buffer (buffering).
(2) If the size of the main heap is n', the size of the insertion buffer is $O(\lg^\alpha n')$ for a fixed integer $\alpha \geq 2$.

(3) When the insertion buffer reaches its maximum size, it is submerged into the main heap in one go (partial rebuilding).
(4) When an *extract-min* operation is performed, the element at the last array position is used as a replacement for the element being removed. This replacement can come either from the insertion buffer or from the main heap (if the insertion buffer is empty).

In the basic form, we implement the main heap as a binary heap and the insertion buffer as a multiary heap. Depending on the arity of the multiary heap (d) and the maximum capacity set for the insertion buffer (b), the performance characteristics of the priority-queue operations follow.

A pseudo-code description of the data structure is given in Fig. 6. Its core is the *bulk-insert* procedure which operates as Floyd's heap-construction algorithm [26], but it only visits the ancestors of the nodes of the insertion buffer, not all the nodes.

```
class lazy-heap

private

data n′: index, n: index,
     a: element[ ]
     min: index
     b: index, d: arity,
     buffer: multiary-heap

procedure bulk-insert
input n′: index, n: index
r ← n − 1
ℓ ← max{n′, parent(r) + 1}
while r ≠ 0:
    r ← parent(r)
    ℓ ← parent(ℓ)
    for i ← r down to ℓ:
        sift-down(i, n)

procedure update-min
input x: element
a[0] ← x
sift-down(0, n′)

public

procedure minimum
output element
assert n > 0
return a[min]

procedure insert
input x: element
if n′ + b ≤ n:
    bulk-insert(n′, n)
    n′ ← n
    min ← 0
    d ← ⌊(1/3) lg n⌋
    b ← d^3
    buffer.construct(0, d, a + n′)
buffer.insert(x)
n ← n + 1
if n′ < n and a[n′] < a[min]:
    min ← n′

procedure extract-min
assert n > 0
n ← n − 1
if min = n′:
    buffer.extract-min()
else:
    update-min(a[n])
if n′ ≥ n or a[0] < a[n′]:
    min ← 0
else:
    min ← n′
```

Fig. 6 Buffering in a lazy heap; the main heap is organized as a binary heap in $\mathbf{a}[0\colon n')$ and the insertion buffer as a multiary heap in $\mathbf{a}[n'\colon n)$; since the third parameter of *buffer.construct* is passed by reference, the changes made by *buffer.insert* and *buffer.extract-min* are visible here and, if the buffer is non-empty, $\mathbf{a}[n']$ is its minimum

As earlier, the ancestors are visited bottom up, level by level, by calling the *sift-down* procedure for each node. The key observation is that, as long as there are more than two nodes considered at a level, the number of visited nodes almost halves at the next level.

In the following analysis we separately consider two phases of the *bulk-insert* procedure. The first phase comprises the *sift-down* calls for the nodes at the levels with more than two involved nodes. Recall that the size of the initial bulk is b. The number of the nodes visited at the jth last level is at most $\lfloor (b-2)/2^{j-1} \rfloor + 2$. For a node at the jth last level, a call to the *sift-down* procedure requires $O(j)$ work. In the first phase, the amount of work involved is $O(\sum_{j=2}^{\lceil \lg n' \rceil} j/2^{j-1} \cdot b) = O(b)$. The second phase comprises at most $2\lfloor \lg n \rfloor$ calls to the *sift-down* subroutine; this accounts for a total of $O(\lg^2 n)$ work.

In the first phase of *bulk-insert* at most $4b + o(b)$ element comparisons are performed (in principle, this is Floyd's heap construction for a subheap of size $2b+o(b)$), and in the second phase of *bulk-insert* at most $2\lg^2 n$ element comparisons are performed (two *sift-down* calls on each level i of the heap, each using at most $2i$ element comparisons). It follows that the amortized number of element comparisons for the bulk insertion is $4 + (2\lg^2 n)/b + o(1)$ per *insert*. To maintain the cursor to the overall minimum, one additional element comparison per *insert* is necessary. The amortized cost per *insert* is then $5 + (2\lg^2 n)/b + o(1)$ element comparisons in addition to the element comparisons performed when inserting an element into the buffer.

Let us consider the parameter values $d \stackrel{\text{def}}{=} \lfloor \lg n \rfloor$ and $b \stackrel{\text{def}}{=} d^2$. In this case the d-ary heap can have up to 3 levels. Therefore, inside the buffer, each *insert* requires at most 2 element comparisons and each *extract-min* at most $2\lg n$ element comparisons. When the insertion costs are amortized over *insert* operations, the amortized number of element comparisons performed per *insert* is $9 + o(1)$. By setting the arity of the multiary heap down to $d \stackrel{\text{def}}{=} \lfloor (1/3)\lg n \rfloor$ and by increasing the maximum capacity of the insertion buffer to $b \stackrel{\text{def}}{=} d^3$, i.e. the d-ary heap would have at most 4 levels, the number of element comparisons per *extract-min* would reduce to $\lg n + O(1)$ if the removed element is inside the buffer. With these parameter values, the cost associated with the second phase of *bulk-insert* would become asymptotically insignificant and the amortized number of element comparisons per *insert* would reduce to $8 + o(1)$.

By applying the stopover optimization for *extract-min* [7, 27], the number of element comparisons per *extract-min* could be reduced to $\lg n + \log^* n + O(1)$ if the removed element is inside the main heap.

To summarize, *construct* could use the best in-place algorithm known for the construction of a binary heap [10], *minimum* reports the overall minimum to which a cursor is maintained, *insert* is delegated to the insertion buffer which is occasionally submerged into the main heap, and *extract-min* is delegated to the component where the overall minimum resides.

Theorem 2 *Let n denote the number of elements stored in a data structure. A lazy heap is an in-place priority queue, for which*

(1) *construct requires* $O(n)$ *worst-case time performing at most* $(13/8)n + o(n)$ *element comparisons;*
(2) *minimum requires* $O(1)$ *worst-case time involving no element comparisons;*
(3) *insert requires* $O(1)$ *time involving* $8 + o(1)$ *element comparisons in the amortized sense; and*
(4) *extract-min requires* $O(\lg n)$ *worst-case time involving at most* $\lg n + \log^* n + O(1)$ *element comparisons.*

6 Relaxed Lazy Heaps: Deamortizing Bulk Insertions

In this section we show how a lazy heap can be modified to execute *insert* in $O(1)$ worst-case time, still supporting the other operations with the same efficiency as before. The action plan can be briefly stated as follows. Instead of performing a bulk insertion in one go, it is executed incrementally by dividing the work among several *insert* and *extract-min* operations such that the work done per operation only increases by a constant (deamortization). More specifically, when the insertion buffer becomes full, it will be closed for insertions and further insertions will be directed to a new insertion buffer, while concomitantly the old buffer is submerged into the main heap in an incremental fashion. We call the incremental process a *submersion* to distinguish it from the bulk insertion. Obviously, such a submersion should be done fast enough to complete before the insertion buffer becomes full again.

Already from this initial description it is clear that the desired data structure, that we call a *relaxed lazy heap*, is composed of three parts: *main heap*, *submersion area*, and *insertion buffer*. The main heap including the submersion area is laid out in an array as a binary heap, and the insertion buffer occupies the last array locations. When a submersion starts, the insertion buffer forms an embryo of the submersion area which lies between the main heap and the new—at this point empty—insertion buffer. When the submersion progresses and *sift-down* is called for each of the ancestors of the initial embryo, the submersion area becomes bigger and bigger until the last *sift-down* for the root of the main heap finishes, and the submersion completes.

Since we do not aim at optimizing the additive constant terms, we can restrict the maximum size of the insertion buffer to be $O(\lg^2 n)$. From this, it follows that the maximum size of the submersion area is also $O(\lg^2 n)$. When the submersion is in progress, after performing *sift-down* for some nodes, it might happen that the overall minimum is in the submersion area. To find the new minimum, we cannot scan the whole submersion area, but we have to maintain a multiary-heap-like data structure to support fast minimum finding. Also, since the insertion buffer is all of a sudden made a submersion area, the structure maintained in both should be the same.

The following list summarizes the rules of the game:

(1) *insert* adds every new element to the insertion buffer.
(2) When the insertion buffer reaches its maximum size, it is made an embryo for a new submersion area and a new empty insertion buffer is created.
(3) If the size of the main heap is n' when a submersion area was created the last time, the size of the insertion buffer is $O(\lg^2 n')$.

(4) The submersion area is incrementally submerged into the main heap by performing a constant amount of work in connection with every modifying operation (*insert* or *extract-min*).
(5) When the insertion buffer is full again, the incremental submersion must have been completed.
(6) When an *extract-min* is performed, the element at the last array position is used as a replacement for the element being removed. This replacement can come from the insertion buffer, from the submersion area (if the insertion buffer is empty), or from the main heap (if both the insertion buffer and the submersion area are empty).

The insertion buffer should support *insert* and *extract-min* efficiently, the submersion area should support *extract-min* efficiently, and the main heap should support *construct* and *extract-min* efficiently. Next we consider, component by component, how to provide these operations.

In the life cycle of the data structure, let us consider the moments when a new insertion buffer is created. Let n' denote the size of the main heap just before this happened the last time. As before, we organize the insertion buffer as a multiary heap, but now we set the arity to $d \stackrel{\text{def}}{=} \lfloor (1/4) \lg n' \rfloor$ and the maximum capacity to d^2. A new element can be added to the end of the buffer and thereafter at most $O(1)$ work is necessary to reestablish the d-ary heap property. When the minimum of the buffer at its first location is to be removed, the last element (if any) is moved into its place. The following *sift-down* operation involves at most d element comparisons per level up to two levels and only a constant number of element moves are needed to reestablish the d-ary heap property. Here, the number of element comparisons performed per *extract-min* is at most $(1/2) \lg n + O(1)$. When a replacement is needed for the old minimum, it is never a problem to use the last element of the insertion buffer for this purpose.

To track the progress of the submersion process, we maintain two intervals that represent the nodes up to which the *sift-down* subroutine has been called. Each such interval is represented by two indices indicating its left and right endpoints, denote them $[\ell_1 : r_1)$ and $[\ell_2 : r_2)$. These two intervals are at consecutive levels, and the parent of the right endpoint of the first interval is the left endpoint of the second interval, i.e. $\ell_2 = \mathit{parent}(r_1)$. We say that these two intervals form the *frontier*. The *submersion area* thus comprises the initial embryo plus the ancestors up to the frontier. The frontier imparts that a *sift-down* is being performed starting from the node whose index is ℓ_2. Once *sift-down* returns, the frontier is updated. While the process advances, the frontier moves upwards and shrinks until it has one or two nodes. When the frontier passes the root, the incremental submersion is complete.

We keep the elements on the frontier in two multiary heaps (heaps in heaps). The elements in the interval $[\ell_1 : r_1)$ are organized as a normal multiary heap and the elements in the interval $[\ell_2 : r_2)$ as a mirrored multiary heap (mirroring). Hence, the minimum of the frontier is at either of its two ends, $a[\ell_1]$ or $a[r_2 - 1]$. Initially, the insertion buffer is made the multiary heap of the interval $[\ell_1 : r_1)$. The arity of the heaps is inherited from the insertion buffer, from which this initial embryo was created. When the upper interval is extended by one position, we have to call *sift-down* at position ℓ_2. Thereafter, we have to *insert* the new element at that position to

the mirrored multiary heap. As the result of this *insert*, the position ℓ_2 may get a new larger element, so we have to invoke *sift-down* at this position one more time. This can be repeated for at most two interleaved calls to *sift-down* in the main heap followed by *sift-up* in the mirrored multiary heap. At the next round, when the interval $[\ell_1 : r_1)$ vanishes, the scanning direction must be reversed so that a mirrored multiary heap is used to fill a normal multiary heap, and so on. After $2 \lg \lg n' + O(1)$ rounds, the multiary heaps shrink to at most two elements, and a single cursor to the minimum can be maintained.

To summarize, the information being maintained to record the state of the submersion process is two intervals of indices to represent the frontier, the variables required by the two multiary heaps, the node that is under consideration by the current *sift-down*, and an activation record telling which of the two *sift-down* operations is in progress at the current node. In spite of the two *sift-down* calls per node, the worst-case running time of the whole process is still linear in the size of the initial embryo. In the actual implementation of the submersion, we could count the number of levels processed by the *sift-down* calls to ensure that we make progress in the desired speed.

To remove the minimum of the submersion area, we know that it must be on the frontier and we readily have its index. This minimum is replaced with the last element of the array. In the case that there are at most two nodes on the frontier, we just perform a *sift-down*, update the cursor to the minimum of the frontier, and we are done. In the case that there are more than two nodes on the frontier, first a *sift-down* in the multiarray heap is called for the node on the frontier that got a replacement. As a result, at most three nodes among the nodes of the frontier may get new replacement elements. For each of these nodes a *sift-down* operation in the main heap may be necessary to remedy the order between the replacement elements and the elements in their descendants. Luckily, the height of the nodes on the frontier is at most $2 \lg \lg n + O(1)$ so we can afford to do these operations. To sum up, extracting the minimum of the submersion area requires $O(\lg n)$ time with at most $(1/2) \lg n + O(\lg \lg n)$ element comparisons.

When we extract the last element of the submersion area, we want to be sure that there is no unwanted interference with the other operations (concurrency management). Therefore, one caution we have to take is that, in submersion, *sift-down* in progress should never stop at a leaf. This way, when we take a replacement from the submersion area, we will not lose the position of this incremental process.

In the main heap, the *minimum* and *extract-min* operations are performed as in a binary heap with the same cost limitations. An exception is that, if *extract-min* meets the frontier of the submersion area, we stop the execution before crossing it. Still, in *extract-min*, the optimization with occasional stopovers [7, 27] could be used, if wanted.

Since a binary heap would be a legal relaxed lazy heap having no submersion area or insertion buffer, *construct* could use the best in-place algorithm known for the construction of a binary heap [10]. By maintaining a cursor to the overall minimum, *minimum* can be trivially supported in $O(1)$ worst-case time and no element comparisons are needed. And, as we have seen, the insertion buffer can support *insert* efficiently and the work involved in the submersion can be split over several modifying operations. All three components support *extract-min* in $O(\lg n)$ worst-case

time; as to the number of element comparisons, *extract-min* in the main heap is the bottleneck.

Theorem 3 *Let n denote the number of elements stored in a data structure. A relaxed lazy heap is an in-place priority queue for which*

(1) *construct requires $O(n)$ worst-case time performing at most $(13/8)n + o(n)$ element comparisons;*
(2) *minimum and insert require $O(1)$ worst-case time; and*
(3) *extract-min requires $O(\lg n)$ worst-case time involving at most $\lg n + \log^* n + O(1)$ element comparisons.*

7 Strengthened Lazy Heaps: Putting Things Together

Our final construction is similar to the one of the previous section in that there are three components: main heap, submersion area, and insertion buffer. Here the main heap has two layers: a *top heap* that is a binary heap, and each leaf of the top heap roots a *bottom heap* that is a complete strong heap (stratification). The main heap is laid out in the array as a binary heap and, in accordance, every bottom heap is scattered throughout the array. As before, the submersion area is inside the main heap, leading to a possible disobedience of heap order at its frontier. Because the main heap is only partially strong, we call the resulting data structure a *strengthened lazy heap*. To help the reader get a complete picture of the data structure, we visualize it in Fig. 7.

The main new ingredient is the *border* maintained between the top heap and the bottom heaps (for an overview of the data maintained, see Fig. 8). When the data structure contains n elements, the *target height* of the top heap is set to $\lceil \lg n - \lg \lg n \rceil$.

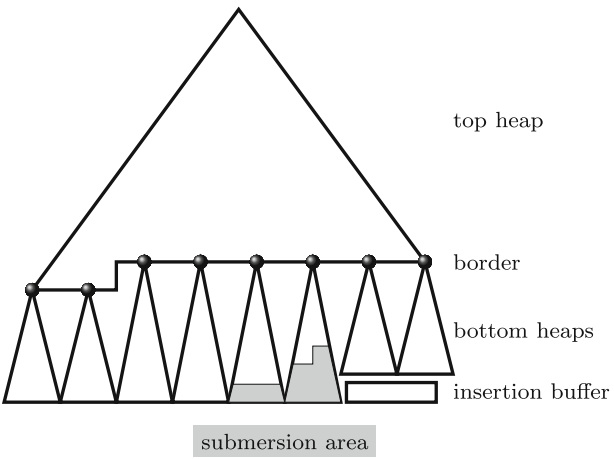

Fig. 7 Schematic view of a strengthened lazy heap

```
class strengthened-lazy-heap

private

data n'': index, n': index, n: index
    a: element[ ]
    min: index
    h_0: height
    border: struct {
        ℓ_1: index, r_1: index
        ℓ_2: index, r_2: index
    }
    frontier: struct {
        ℓ_0: index, r_0: index
        ℓ_1: index, r_1: index
        ℓ_2: index, r_2: index
        ℓ_3: index, r_3: index
    }
    d': index
    I_0: multiary-heap
    locked-tree-min: index
    processed-roots-min: index
    frontier-min: index
    d: index
    buffer: multiary-heap
```

Fig. 8 Most important variables of a strengthened lazy heap; the main heap is held in $\mathbf{a}[0 : n'')$, the embryo of the submersion area in $\mathbf{a}[n'' : n')$, and the insertion buffer in $\mathbf{a}[n' : n)$

Since n varies and the target height is a function of n, the *actual height* (denote it h_0) cannot exactly match this bound, but the goal is to keep it close to the target height up to an additive constant. Since the border must be moved dynamically depending on the changes made to n, we represent it in the same way as the frontier in Section 6 using two intervals $[\ell_1 : r_1)$ and $[\ell_2 : r_2)$. To lower the border, we just adjust h_0 and ignore the left-dominance property for the nodes on the previous border. To raise the border, we need a new incremental remedy process that scans the nodes on the old border and applies *strengthening-sift-down* on each left child (see the procedure *raise-border* in Fig. 9). Again, we only need a constant amount of space to record the state of this process.

The work to be done in border raising can be scheduled as follows. In connection with every modifying operation, after accomplishing its real task, if $n = 2^h$ for some positive integer h, we check the relationship between $\lceil h - \lg h \rceil$ and h_0. If they are equal, there is no need to move the border. If necessary, border lowering can be done immediately since it only involves $O(1)$ work. Otherwise, we initiate an incremental border-raising process (deamortization). The total work involved in border raising is $O(n)$. We divide this work evenly among the forthcoming 2^{h-1} modifying operations, so that each of them takes a constant share of the work. Therefore, border

```
procedure lower-border
border.ℓ1 ← left-child(border.ℓ1)
border.r1 ← left-child(border.r1)
border.ℓ2 ← border.ℓ1
border.r2 ← border.r1
h0 ← h0 + 1

procedure raise-border
border.r2 ← parent(border.r1)
for i ← border.r1 − 1 down to border.ℓ1:
    if not even(i):
        border.ℓ2 ← parent(i)
        border.r1 ← i − 1
        strengthening-sift-down(i, n)
border.r1 ← border.r2
border.ℓ1 ← border.ℓ2
h0 ← h0 − 1

procedure refill-buffer
input n'': index, n': index
d ← ⌊(1/4) lg n''⌋
buffer.construct(0, d, a + n'')
for i ← n' to min{n', n} − 1:
    buffer.insert(a[i])
```

Fig. 9 Description of the new incremental processes when executed in one go

raising must be complete before n is increased to 2^{h+1} or decreased to 2^{h-1}, i.e. when the next check is due.

In all essentials, the insertion buffer and the submersion area are implemented as explained before. Let n'' be the size of the main heap just before the latest submersion process was initiated. Further, let n' be its size just after this, i.e. including the size of the embryo of the submersion area. Again, the arity of the multiary heap used in the insertion buffer is set to $d \stackrel{\text{def}}{=} \lfloor (1/4) \lg n' \rfloor$ and the maximum capacity of the buffer is set to d^2. When an insertion buffer is used to create a submersion area, the very same arity is used there. In particular, the arities in the current submersion area (d' in Fig. 8) and the current insertion buffer need not be the same, but their difference is a small constant.

To improve the performance of *extract-min* in the main heap, we use a new procedure, that we call *combined-sift-down* (Fig. 10), instead of *sift-down*. Assume we have to replace the minimum of the main heap with another element. To reestablish heap order, we apply the stopover optimization proposed by Carlsson [6] (see also [7, 27]): We traverse down along the special path until we reach the root of a bottom heap. By comparing the replacement element with the element at that root, we decide whether the replacement element should land in the top heap or in the bottom heap. In the first case, in *binary-search-sift-up*, we find the position of the replacement element using binary search on the traversed path and then do the required element moves. In the second case, we move the elements of the special path one node upwards vacating the root of the bottom tree, then move the replacement element to the root of the bottom tree and apply *swapping-sift-down* on this bottom tree.

Let us now recap how the operations are executed and analyse their performance. Here we ignore the extra work done due to the incremental processes. Since a strong heap would be a legal strengthened lazy heap having no submersion area or insertion buffer, *construct* could use the faster of the two algorithms developed in Section 4. As previously, *minimum* can be carried out in $O(1)$ worst-case time without any element comparisons by maintaining a cursor to the overall minimum and letting the

other operations update this cursor. Also, as before, *insert* adds the given element to the insertion buffer and updates the cursors maintained if necessary. The minimum can be at the root of the top heap, at the first location of the insertion buffer, or at some location on the frontier of the submersion area specified by a separate cursor. In *extract-min*, the minimum could be in any of these three components.

If the minimum is at the root of the top heap, we find a replacement element and apply *combined-sift-down* for the root while making sure not to cross the frontier of the submersion area. The top heap is of size $O(n/\lg n)$ and each of the bottom heaps is of size $O(\lg n)$. To reach the root of a bottom heap, we perform $\lg n - \lg\lg n + O(1)$ element comparisons. If we have to go upwards, we perform $\lg\lg n + O(1)$ additional element comparisons in the binary search while applying the *binary-search-sift-up* operation. On the other hand, if we have to go downwards, *swapping-sift-down* needs to perform at most $\lg\lg n + O(1)$ element comparisons. In both cases, the number of element comparisons performed is at most $\lg n + O(1)$ and the work done is $O(\lg n)$.

If the overall minimum is in the insertion buffer, it is removed as explained in the previous section. This removal involves $2d + O(1)$ element comparisons and the amount of work done is proportional to that number. Since $d = \lfloor (1/4)\lg n' \rfloor$, this operation requires at most $(1/2)\lg n + O(1)$ element comparisons and $O(\lg n)$ work.

If the frontier contains the overall minimum, we apply a similar treatment to that explained in the previous section with a basic exception. If there are more than two nodes on the frontier, the height of the nodes on the frontier is at most $2\lg\lg n + O(1)$. In this case, we use the *strengthening-sift-down* procedure in place of the

```
procedure ancestor
input i: index, h: height
output index
return ⌊(i + 1)/2^h⌋ − 1

procedure rotate
input i: index, k: index, h: height
x ← a[i]
for h′ ← h − 1 down to 0:
    a[ancestor(k, h′ + 1)] ← a[ancestor(k, h′)]
a[k] ← x

procedure correct-place
input i: index, k: index, h: height
output index
h″ ← h
while i ≠ k:
    h′ ← ⌊(h + 1)/2⌋
    j ← ancestor(k, h′)
    h ← h − h′
    if a[j] < a[i]:
        i ← ancestor(k, h)
    else:
        k ← j
        h″ ← h″ − h′
return (i, h″)

procedure binary-search-sift-up
input i: index, k: index, h: height
(j, d) ← correct-place(i, k, h)
rotate(i, j, d)

procedure is-on-border
input i: index
output Boolean
if border.ℓ1 ≤ i and i ≤ border.r1:
    return true
if border.ℓ2 ≤ i and i ≤ border.r2:
    return true
return false

procedure combined-sift-down
input i: index, n: index
j ← i
h ← 0
while not is-on-border(j):
    k ← left-child(j)
    if a[sibling(k)] < a[k]:
        k ← sibling(k)
    j ← k
    h ← h + 1
if a[j] < a[i]:
    rotate(i, j, h)
    swapping-sift-down(j, n)
else:
    binary-search-sift-up(i, j, h)
```

Fig. 10 Implementation of *combined-sift-down*

sift-down procedure. The whole process requires at most $(1/2)\lg n + O(\lg\lg n)$ element comparisons and $O(\lg n)$ work. If there are at most two nodes on the frontier, the frontier lies in the top heap. In this case, we apply the *combined-sift-down* procedure instead. This requires at most $\lg n + O(1)$ element comparisons and $O(\lg n)$ work. Either way, for large enough n, the minimum extraction here requires at most $\lg n + O(1)$ element comparisons.

One important detail that causes some troubles is the fact that the bottom heaps have to be complete. The following two issues pop up:

(1) When we create a submersion area, we cannot use the whole insertion buffer as its embryo. We have to leave the last elements that cannot fill a whole level of a bottom heap in the insertion buffer. We call these elements *leftovers*. After moving most of the elements to the embryo, we do not know the minimum of the leftovers. There may only be a few elements in the submersion area that are smaller than the minimum among the leftovers, but there are definitely some. Hence, we have to find the minimum of the leftovers incrementally if extractions are from the main heap, but we must finish this incremental process in connection with the first extraction from the submersion area. As our forthcoming analysis will show, we have plenty of time for that.

(2) When we need a replacement element for *extract-min*, we cannot take one from the main heap or the submersion area if the insertion buffer is empty. Instead, we have to return a whole level of a bottom heap to the insertion buffer (see the procedure *refill-buffer* in Fig. 9). Conveniently, these elements appear at the rear of the array. Again the problem is that, after this procedure, the minimum of the piece cut would not be known. Fortunately, we can handle these refilling elements almost the same way as the leftovers. The work to be done in minimum finding can be divided evenly among the next $\lfloor(1/4)\lg n\rfloor$ *extract-min* operations if the forthcoming extractions are from the main heap or if they are from the submersion area when the frontier is on or above level $\lfloor(1/2)\lg n\rfloor$. The correctness follows from the fact that, in this case, there is a logarithmic number of elements that are smaller than any of the refilling elements. Alternatively, as the forthcoming analysis will show, the work can be finished in one go if an extraction is from the submersion area when the frontier is below level $\lfloor(1/2)\lg n\rfloor$.

The special case, where it is necessary to complete an incremental buffer-refilling process in one go following an extraction, will increase the number of element comparisons performed by at most $(1/4)\lg n + O(1)$. First, if the submersion process is in its first phase when there are more than two nodes on the frontier, the number of element comparisons will increase to at most $(3/4)\lg n + O(\lg\lg n)$. This case also applies for leftovers. Second, if the submersion process is below level $\lfloor(1/2)\lg n\rfloor$, but is in its second phase when there are at most two nodes on the frontier, *combined-sift-down* will only require at most $(1/2)\lg n + O(1)$ element comparisons, and the total will be at most $(3/4)\lg n + O(1)$ element comparisons. In both cases, for large enough n, the number of element comparisons performed is still bounded by $\lg n + O(1)$.

If we swapped two subtrees in the bottom tree where the frontier consists of two intervals, there is a risk that we mess up the frontier. In accordance, as a change of plans, we schedule the submersion process differently: We process the bottom trees one by one, and lock the bottom tree under consideration to skip subtree interchanges initiated by *extract-min* in the main heap (concurrency management). Then, the frontier may be divided into four intervals (see Fig. 8):

(1) the interval corresponding to the unprocessed leaves of the initial embryo (interval $[\ell_0 : r_0)$; initially, $\ell_0 = n''$ and $r_0 = n'$),
(2) the two intervals $[\ell_1 : r_1)$ and $[\ell_2 : r_2)$ in the bottom tree under consideration, and
(3) the interval of the roots of the bottom heaps that have been handled by the submersion process (interval $[\ell_3 : r_3)$).

Locking resolves the potential conflict with *extract-min*. However, in the currently processed bottom tree there are some nodes between the root and the frontier that are not yet included in the submersion process and are not in order with the elements above or below. This is not a problem, as none of these elements can be the overall minimum except after a logarithmic number of modifying operations. Within such time, these nodes should have already been handled by the submersion process.

Inside the initial embryo, a multiary heap (I_0 in Fig. 8) is used to keep track of the minimum of these elements. Since the processed bottom tree is so small, a separate cursor can be maintained to point to the minimum on this part of the frontier. Also, the last interval will never grow larger than $d + O(1)$, so a cursor to the minimum stored at these nodes will also do here.

As shown, due to the two-layer structure and the completeness requirement for the bottom heaps, the incremental remedy processes are more complicated for a strengthened lazy heap than for a relaxed lazy heap. Naturally, we want to avoid any undesirable interference between the incremental processes and the priority-queue operations. Last, let us consider the introduced complications one at a time and sketch how we handle them (concurrency management).

(1) To avoid interference between *refill-buffer* and *strengthening-sift-down*, an incremental process should never stop at a leaf. One way to enforce this in the submersion area is to keep every sibling pair of elements in sorted order when they are inserted into the buffer. Then *strengthening-sift-down* need never be called for several leaves after each other.
(2) If *extract-min* meets the node where border-raising *strengthening-sift-down* is to be applied, we stop the execution of *extract-min* and let the incremental process reestablish strong heap order below that node.
(3) If the node where border-raising *strengthening-sift-down* is to be applied is inside the submersion area, we stop this corrective action and jump to the next since the submersion process must have already established strong heap order below that node.
(4) If the node processed by submersion *strengthening-sift-down* and that by border-raising *strengthening-sift-down* meet, we stop the border-raising process

and jump to the next since the submersion process will reestablish strong heap order below that node.

(5) If the border-raising *strengthening-sift-down* meets the frontier, we stop this corrective action before crossing it and jump to the next.

(6) When the node recorded by border-raising *strengthening-sift-down* is moved by a *swapping-sift-down*, its index is to be updated accordingly.

Because of the subtree interchanges made in *swapping-sift-down*, the number of element moves performed by *extract-min*—even though asymptotically logarithmic—would be larger than the number of element comparisons. However, we can control the number of element moves by making the bottom heaps smaller and applying the stopover optimization with several stops before using *swapping-sift-down*. For a fixed integer $\gamma \geq 1$, we could limit the size of the bottom heaps to $O(\lg^{(\gamma)} n)$ (the logarithm taken γ times), after which the number of element moves would be upper bounded by $\lg n + O(\lg^{(\gamma)} n)$, while the bounds for the other operations still hold.

The description and analysis of the data structure ends here. Thus, we have proved our main result.

Theorem 4 *Let n denote the number of elements stored in a data structure. A strengthened lazy heap is an in-place priority queue, for which*

(1) *construct requires $O(n)$ worst-case time performing at most $(23/12)n + o(n)$ element comparisons;*

(2) *minimum and insert require $O(1)$ worst-case time; and*

(3) *extract-min requires $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons.*

8 Conclusions

In this paper we described a priority queue that

(1) operates in-place;

(2) supports *construct*, *minimum*, *insert*, and *extract-min* in asymptotically optimal worst-case time; and

(3) executes *minimum*, *insert*, and *extract-min* with the minimum number of element comparisons up to additive constant terms.

It is remarkable that we could surpass the lower bounds on the number of element comparisons performed by *insert* and *extract-min* known for binary heaps [27] by slightly loosening the assumptions that are intrinsic to these lower bounds. To achieve our goals, we simultaneously imposed more order on some nodes, by forbidding some elements at left children to be larger than those at their right siblings, and less order on others, by allowing some elements to possibly be smaller than those at the parents.

In retrospect, we admit that, while binary heaps [49] are practically efficient, our data structures are mainly of theoretical value. As a warm-up, we proved that *insert*

can be carried out in amortized $O(1)$ time by performing amortized $8 + o(1)$ element comparisons. For the worst-case constructions, we only proved such a constant exists, but made no attempt to specify its value. We have implemented the amortized solution; this reference implementation was not competitive with binary heaps, but it was not hopelessly slow either.

The main questions left open are

(1) whether the number of element comparisons for *construct* can be lowered;
(2) whether the number of element moves for *extract-min* can be lowered;
(3) whether our constructions could be simplified; and
(4) whether there are components that are useful in practice.

Following preliminary versions of our work [18, 19, 22], two answers to the question about the number of element moves have been given. Brodal et al. [3] showed that, in the amortized sense, asymptotically optimal time bounds and amortized $O(1)$ element moves per *insert* and *extract-min* are achievable by an in-place structure. Darwish et al. [12] proved that asymptotically optimal worst-case bounds involving $O(1)$ element moves per *insert* and *extract-min* are achievable if memory space for $O(n/\lg n)$ additional bits is available.

References

1. Alstrup, S., Husfeldt, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues. ACM Trans. Algorithms **1**(1), 102–106 (2005)
2. Bollobás, B., Fenner, T., Frieze, A.: On the best case of heapsort. J. Algorithms **20**(2), 205–217 (1996)
3. Brodal, G.S., Nielsen, J.S., Truelsen, J.: Strictly implicit priority queues: on the number of moves and worst-case time. In: WADS 2015, vol. 9214, pp. 91–102. Springer, Heidelberg (2015)
4. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM J. Comput. **7**(3), 298–319 (1978)
5. Bruun, A., Edelkamp, S., Katajainen, J., Rasmussen, J.: Policy-based benchmarking of weak heaps and their relatives. In: SEA 2010, LNCS, vol. 6049, pp. 424–435. Springer, Heidelberg (2010)
6. Carlsson, S.: A variant of Heapsort with almost optimal number of comparisons. Inform. Process. Lett. **24**(4), 247–250 (1987)
7. Carlsson, S.: An optimal algorithm for deleting the root of a heap. Inform. Process. Lett. **37**(2), 117–120 (1991)
8. Carlsson, S., Chen, J., Mattsson, C.: Heaps with bits. Theoret. Comput. Sci. **164**(1-2), 1–12 (1996)
9. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: SWAT 1988, LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
10. Chen, J., Edelkamp, S., Elmasry, A., Katajainen, J.: In-place heap construction with optimized comparisons, moves, and cache misses. In: MFCS 2012, LNCS, vol. 7464, pp. 259–270. Springer, Heidelberg (2012)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. Introduction to Algorithms, 3th edn. The MIT Press, Cambridge (2009)
12. Darwish, O., Elmasry, A., Katajainen, J.: Memory-adjustable navigation piles with applications to sorting and convex hulls. E-print arXiv:1510.07185, arXiv.org Ithaca (2015)
13. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM **31**(11), 1343–1354 (1988)
14. Dutton, R.D.: Weak-heap sort. BIT **33**(3), 372–381 (1993)
15. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap data structure: Variants and applications. J. Discrete Algorithms **16**, 187–205 (2012)

16. Edelkamp, S., Elmasry, A., Katajainen, J.: A catalogue of weak-heap programs. CPH STL Report 2012-2. Dept. Comput. Sci., Univ, Copenhagen, Copenhagen (2012)
17. Edelkamp, S., Elmasry, A., Katajainen, J.: Weak heaps engineered. J. Discrete Algorithms **23**, 83–97 (2013)
18. Edelkamp, S., Elmasry, A., Katajainen, J.: Strengthened lazy heaps: Surpassing the lower bounds for binary heaps. E-print arXiv:1407.3377, arXiv.org Ithaca (2014)
19. Edelkamp, S., Elmasry, A., Katajainen, J.: An in-place Priority Queue with $O(1)$ Time for Push and $\lg n + O(1)$ Comparisons for Pop. In: CSR 2015, LNCS, vol. 9139, pp. 1–15. Springer, Heidelberg (2015)
20. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms **5**(1), 14:1–14:19 (2008)
21. Elmasry, A., Jensen, C., Katajainen, J.: Two skew-binary numeral systems and one application. Theory Comput. Syst. **50**(1), 185–211 (2012)
22. Elmasry, A., Katajainen, J.: Towards ultimate binary heaps CPH STL report 2013-1. Dept. Comput. Sci., Univ, Copenhagen, Copenhagen (2013)
23. van Emde Boas, P.: Thirty nine years of stratified trees. In: ISCIM 2013, vol. 1, pp. 1–14. Epoka University, Tirana (2013)
24. Fleischer, R.: A tight lower bound for the worst case of Bottom-up-heapsort. Algorithmica **11**(2), 104–115 (1994)
25. Fleischer, R., Sinha, B., Uhrig, C.: A lower bound for the worst case of Bottom-up-heapsort. Inform. and Comput. **102**(3), 263–279 (1993)
26. Floyd, R.W.: Algorithm 245: Treesort 3. Commun. ACM **7**(12), 701 (1964)
27. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. **15**(4), 964–971 (1986)
28. Han, Y.: Deterministic sorting in $O(n \log\log n)$ time and linear space. J. Algorithms **50**(1), 96–105 (2004)
29. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: FOCS 2002, pp. 135–144. IEEE Computer Society, Los Alamitos (2002)
30. Harvey, N.J.A., Zatloukal, K.C.: The Post-Order Heap. In: FUN 2004 (2004)
31. Hayward, R., McDiarmid, C.: Average case analysis of heap building by repeated insertion. J. Algorithms **12**(1), 126–153 (1991)
32. Johnson, D.B.: Priority queues with update and finding minimum spanning trees. Inform. Process. Lett. **4**(3), 53–57 (1975)
33. Katajainen, J.: The Ultimate Heapsort. In: CATS 1998, Australian computer science communications, vol. 20, pp. 87–95. Springer, Singapore (1998)
34. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms, 3rd edn, vol. 1. Addison Wesley Longman, Reading (1997)
35. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, 2nd edn, vol. 3. Addison Wesley Longman, Reading (1998)
36. Levcopoulos, C., Petersson, O.: Adaptive heapsort. J. Algorithms **14**(3), 395–413 (1993)
37. McDiarmid, C.J.H., Reed, B.A.: Building heaps fast. J. Algorithms **10**(3), 352–365 (1989)
38. Nievergelt, J., Reingold, E.: Binary search trees of bounded balance. SIAM J. Comput. **2**(1), 33–43 (1973)
39. Overmars, M.H.: The Design of Dynamic Data Structures, LNCS, vol. 156. Springer, Heidelberg (1983)
40. Sack, J.R., Strothotte, T.: A characterization of heaps and its applications **86**(1), 69–86 (1990)
41. Schaffer, R., Sedgewick, R.: The analysis of heapsort. J. Algorithms **15**(1), 76–100 (1993)
42. Suchenek, M.A.: Elementary yet precise worst-case analysis of Floyd's heap-construction program. Fund. Inform. **120**(1), 75–92 (2012)
43. Thorup, M.: Equivalence between priority queues and sorting. J. ACM **54**(6), 28:1–28:27 (2007)
44. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM **21**(4), 309–315 (1978)
45. Wang, X., Wu, Y., Zhu, D.: A new variant of in-place sort algorithm. Procedia Eng. **29**, 2274–2278 (2012)
46. Wegener, I.: The worst case complexity of McDiarmid and Reed's variant of Bottom-up Heapsort is less than $n \log n + 1.1n$. Inform. and Comput. **97**(1), 86–96 (1992)
47. Wegener, I.: Bottom-up-heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoret. Comput. Sci. **118**(1), 81–98 (1993)

48. Wegener, I.: A simple modification of Xunrang and Yuzhang's Heapsort variant improving its complexity significantly. Comput. J. **36**(3), 286–288 (1993)
49. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM **7**(6), 347–348 (1964)
50. Xunrang, G., Yuzhang, Z.: A new Heapsort algorithm and the analysis of its complexity. Comput. J. **33**(3), 281–282 (1990)
51. Xunrang, G., Yuzhang, Z.: Asymptotic optimal Heapsort algorithm. Theoret. Comput. Sci. **134**(2), 559–565 (1994)
52. Xunrang, G., Yuzhang, Z.: Optimal heapsort algorithm. Theoret. Comput. Sci. **163**(1–2), 239–243 (1996)